

Graphical Java Lesson 1-8: Making Mario Jump

In the last lesson, we made Mario fall under the influence of gravity. He was able to land on a platform, and walk on it. But what would a Mario game be without jumping? In this lesson, we're going to give Mario the ability to jump, and we're going to give our world some more platforms for him to jump on.

In Super Mario Bros., there is no button that causes Mario to fall. However, when the player presses the A button, on his controller, Mario jumps, if possible. If the player presses it for a short time, Mario doesn't jump very high. If the player holds it down for a little longer, Mario jumps higher. Let's add a "jump" button to our virtual controller. Go to your `Controller.java` source file. Add the following member variable to your `Controller` class:

```
private boolean jump;
```

The variable will be of type `boolean`, because the "jump" button can either be "down" or "not down". We'll worry about how long it's been down later. Initially, the "jump" button, just like the other buttons, will not be down, so add the following line of code to the constructor of your `Controller` class:

```
jump = false;
```

We'll also add the trio of methods that will press the "jump" button, release it, and check whether it's down or not. Add the following methods to your `Controller` class:

```
public void pressJump()
{
    jump = true;
}
```

```
public void releaseJump()
{
    jump = false;
}
```

```
public boolean isJumpButtonDown()
{
    return jump;
}
```

The "jump" button can be down while the "left" or "right" button is down, so there is no need to mess with the states of either of those two buttons, in our `pressJump` method. Finally, whenever we pause the game, we want to cause the "jump" button to be released (just like the other buttons), and "lock the player out of it", during that time. Add the following line of code to your `releaseAll` method:

```
jump = false;
```

Now, we'll connect the state of the "jump" button to the player's pressing and letting go of the A key. Go to your `KBInputProcessor.java` source file. In the `keyPressed` method of your `KBInputProcessor` class, add the following block of code to your if statement:

```
else if (key_code == KeyEvent.VK_A)
{
    if (!WalkingMario.paused)
        WalkingMario.ctrl.pressJump();
}
```

If the player presses the A key, and the game is not paused, the "jump" button will be pressed. When the player lets go of the A key, we want the "jump" button to be released. In the `keyReleased` method of your `KBInputProcessor` class, add the following block of code to your if statement:

```
else if (key_code == KeyEvent.VK_A)
    WalkingMario.ctrl.releaseJump();
```

By the way, if you don't want to use the A key, to make Mario jump, you can substitute the virtual key code of the key you want to use, in place of `VK_A`, in BOTH of these blocks. **Students probably have the Table of Virtual Key Codes.**

Now that our controller has a "jump" button, we want to make Mario jump, whenever it's pressed. Go to your `Mario.java` source file. Mario currently has his own gravity setting, which specifies how fast he falls, when he's falling. Now, we'll give Mario his own "jump strength" setting, which will specify how fast he rises, when he's jumping. We'll also need to give him a piece of state that will specify whether he's in the process of jumping or not. If he's not in the process of jumping, he'll "attempt to" fall. Add the following member variables to your `Mario` class:

```
private int jump_strength;
private boolean jumping;
```

`jumping` will only be `true`, when Mario is rising, during a jump. At all other times, it will be `false`. To initialize these variables, add the following code to the constructor of your `Mario` class:

```
jump_strength = 5;
jumping = false;
```

What does a jump strength of "5" mean? What units are we using here? This will work just like Mario's gravity setting. Every 20 milliseconds (one tick of our timer), if Mario is jumping, he will attempt to rise 5 pixels. Therefore, Mario's jump strength will also be measured in pixels per tick.

Now that we've given our controller a "jump" button, and specified Mario's jump strength, we want to make Mario respond to presses of the "jump" button, by jumping. A jump can be broken down into three parts:

- Its beginning: the first tick after the "jump" button is pressed.
- Its middle: the ticks from the second tick after the "jump" button is pressed to the last tick before the "jump" button is released.
- Its end: the first tick after the "jump" button is released.

For each of these parts, we'll add a method to the `Mario` class. We'll make sure that method is called, when that part is supposed to take place. First, let's specify what should take place when Mario begins to jump. Two things should happen:

- He should go from "not jumping" to "jumping".
- He should attempt to rise a number of pixels equal to his jump strength.

Add the following method to your `Mario` class:

```
public void startJump()  
{  
    jumping = true;  
    y_velocity = -jump_strength;  
}
```

We set the *y* component of Mario's velocity to `-jump_strength`, rather than `jump_strength`, because before, we specified Mario's jump strength as a positive number. Negating a positive number makes it negative. A negative number, in the *y* direction, means "up". Next, we'll specify what should take place while Mario is jumping (the "middle" of his jump, above):

- He should attempt to rise a number of pixels equal to his jump strength.

Add this simple method to your `Mario` class:

```
public void continueJump()  
{  
    y_velocity = -jump_strength;  
}
```

We don't need to set `jumping` to `true` again, because we'll only call `continueJump`, when Mario is already in the process of jumping (when `jumping` is `true`). We will have previously called `startJump` (which set `jumping` to `true`). Finally, we'll specify what should happen right after the player releases the "jump" button (the "end" of Mario's jump, above):

- He should go from "jumping" to "not jumping".
- He should not attempt to rise at all.

Add the following method to your Mario class:

```
public void stopJump()
{
    jumping = false;
    y_velocity = 0;
}
```

Finally, it's time to have Mario's `act` method call these methods, so that he will jump (or not), in response to the player's input. Go to your `act` method. You should see the part that determines whether Mario moves left, moves right, or stands still:

```
if (WalkingMario.ctrl.isLeftButtonDown())
    moveLeft();
else if (WalkingMario.ctrl.isRightButtonDown())
    moveRight();
else
    standStill();
```

You should also see the call to `respondToGravity`:

```
respondToGravity();
```

Between these two parts, insert the following if statement:

```
if (WalkingMario.ctrl.isJumpButtonDown())
{
    if (!jumping)
        startJump();
    else
        continueJump();
}
else if (jumping)
    stopJump();
```

If the "jump" button is down:

- If Mario is not currently jumping, he will begin to jump.
- If Mario is currently in the middle of a jump, he will continue that jump.

If the "jump" button is not down, and Mario was jumping, on the previous tick (that always leaves `jumping` `true`), he will end that jump. If none of the above conditions are true, then Mario won't do anything, with regard to jumping, because the "jump" button is neither down, nor has it recently been released.

The body of your act method should now look like:

```
if (WalkingMario.ctrl.isLeftButtonDown())
    moveLeft();
else if (WalkingMario.ctrl.isRightButtonDown())
    moveRight();
else
    standStill();

if (WalkingMario.ctrl.isJumpButtonDown())
{
    if (!jumping)
        startJump();
    else
        continueJump();
}
else if (jumping)
    stopJump();

respondToGravity();
```

One more thing: we'll need to make a small change to our `respondToGravity` method, so that Mario doesn't "attempt to" fall, while he's jumping. In your `respondToGravity` method, change ...

```
if (!isOnAPlatform())
    y_velocity = gravity;
else
    y_velocity = 0;
```

... to:

```
if (!jumping)
{
    if (!isOnAPlatform())
        y_velocity = gravity;
    else
        y_velocity = 0;
}
```

Essentially, we'll "disable gravity", for Mario, while he's jumping. On any given tick, Mario will either "attempt to" fall or attempt to rise. On NO tick, will he attempt to do both.

Run your program. When you press A, Mario should rise. When you let go of A, Mario should fall, if he's not on top of a platform. This is good, but if you press A, while Mario is in the air, he'll rise from there. Normally, Mario isn't supposed to be able to do that. Also, if you hold down A for a long time, Mario will just keep on rising. He won't reach a maximum height, and then start to fall. This type of control would be great, if we were making a Jetpac game (where your character uses a jet pack, to rise), but we're trying to make a Mario game here. **Try walking off the edge of the platform, and then "jumping" up from the abyss, back on to the platform again. If you are right underneath the platform, can you "jump" up through it, or do you have to go around it?**

Right now, our world only contains one platform. Platformer games are more fun, when there are lots of platforms to jump on. Let's refactor our code, so that our world can contain any number of platforms for Mario to interact with (land on, jump off, be blocked by, etc.). First, go to your `WalkingMario.java` source file. In your list of global variables, you have the following declaration:

```
public static Platform plat;
```

This allows us to use only one `Platform` object. To use any number of `Platforms`, replace that declaration with:

```
public static ArrayList<Platform> plats;
```

Generally, when we want to create a number of objects, and don't know, in advance, how many we want to create, we use an `ArrayList`, to store them. Also, when I'm storing more than one object in a variable, I like to call that variable the plural form of what I'm storing in it (`plats` vs. `plat`). Import the `ArrayList` class, with the following import statement:

```
import java.util.ArrayList;
```

Now that we have our `ArrayList`, we'll need to initialize it, and add some platforms to it. In your main method, replace ...

```
plat = new Platform(new Rectangle(0, WINDOW_HEIGHT - 10,  
    WINDOW_WIDTH, 10), Color.BLACK);
```

... with:

```
plats = new ArrayList<>();  
plats.add(new Platform(new Rectangle(0, WINDOW_HEIGHT - 10,  
    WINDOW_WIDTH, 10), Color.BLACK));
```

Since we're using an `ArrayList` to store them, we can make more than one platform, so let's make two more. Right after the above code, insert the following code:

```
plats.add(new Platform(new Rectangle(100, 210, 200, 10),  
    Color.CYAN));  
plats.add(new Platform(new Rectangle(350, 340, 100, 10),  
    Color.MAGENTA));
```

Now, we have three platforms: a black one, at the bottom of the screen, a cyan one, near the left side and middle of the screen, and a magenta one, closer to the right side and bottom of the screen. That small change to one declaration, in our list of global variables, has "broken" many other parts of our code, such that in its current state, it won't compile.

We are going to refactor our code, to accommodate our world's having more than one platform, so that it will compile again. Go to your `ContentPane.java` source file. In the `paintComponent` method of the `ContentPane` class, we currently have the following line of code draw one platform:

```
WalkingMario.plat.draw(g);
```

To have our program draw all of our platforms, replace it with the following for loop:

```
for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
    WalkingMario.plats.get(i).draw(g);
```

That wasn't so hard. Our `Mario` class will require more work, though. Go to your `Mario.java` source file. Most of the code is okay, but everything that deals with Mario-platform interactions "directly" will need to be refactored. One of the methods we'll need to refactor is `isOnAPlatform`. However, methods that merely call `isOnAPlatform`, and that don't reference the old variable, `plat` (which no longer exists, since we now use `plats`), won't need to be refactored. The algorithms implemented by those methods are solid; only their details, like the number of platforms involved, need to be changed.

We already know what it means for Mario to be on top of a platform. Now, let's change the code of our `isOnAPlatform` method, so that it checks whether Mario is on top of ANY OF our platforms. In the `isOnAPlatform` method of your `Mario` class, replace ...

```
return myBottom() == WalkingMario.plat.myTop() - 1 &&
    horizontalExtent().overlaps(WalkingMario.plat.horizontalExtent());
```

... with the following for loop:

```
for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
{
    Platform plat = WalkingMario.plats.get(i);

    if (myBottom() == plat.myTop() - 1 &&
        horizontalExtent().overlaps(plat.horizontalExtent()))
        return true;
}

return false;
```

Generally, when you refactor a piece of code to use an array or `ArrayList` of objects, instead of a single object, you change a single statement into a for loop. Inside the above for loop, I created a temporary variable, `plat`, to store the current `Platform` object. This was not necessary, but it made the code following that statement easier to read, by allowing us to substitute `plat` for `WalkingMario.plats.get(i)`. If Mario is on top of ANY OF the platforms, this method will return `true`. Else, if Mario is on top of NONE OF the platforms, it will return `false`.

In a similar vein, we can refactor the `isBlockedByALeftWall`, `isBlockedByARightWall`, and `isInAPlatform` methods of the `Mario` class. They also check whether Mario is in a "special" location, relative to that of a platform. In the `isBlockedByALeftWall` method, replace ...

```
return myLeft() == WalkingMario.plat.myRight() + 1 &&
    verticalExtent().overlaps(WalkingMario.plat.verticalExtent());
```

... with the following for loop:

```
for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
{
    Platform plat = WalkingMario.plats.get(i);

    if (myLeft() == plat.myRight() + 1 &&
        verticalExtent().overlaps(plat.verticalExtent()))
        return true;
}

return false;
```

Then, in the `isBlockedByARightWall` method, replace ...

```
return myRight() == WalkingMario.plat.myLeft() - 1 &&
    verticalExtent().overlaps(WalkingMario.plat.verticalExtent());
```

... with this for loop:

```
for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
{
    Platform plat = WalkingMario.plats.get(i);

    if (myRight() == plat.myLeft() - 1 &&
        verticalExtent().overlaps(plat.verticalExtent()))
        return true;
}

return false;
```

Finally, in the `isInAPlatform` method, replace ...

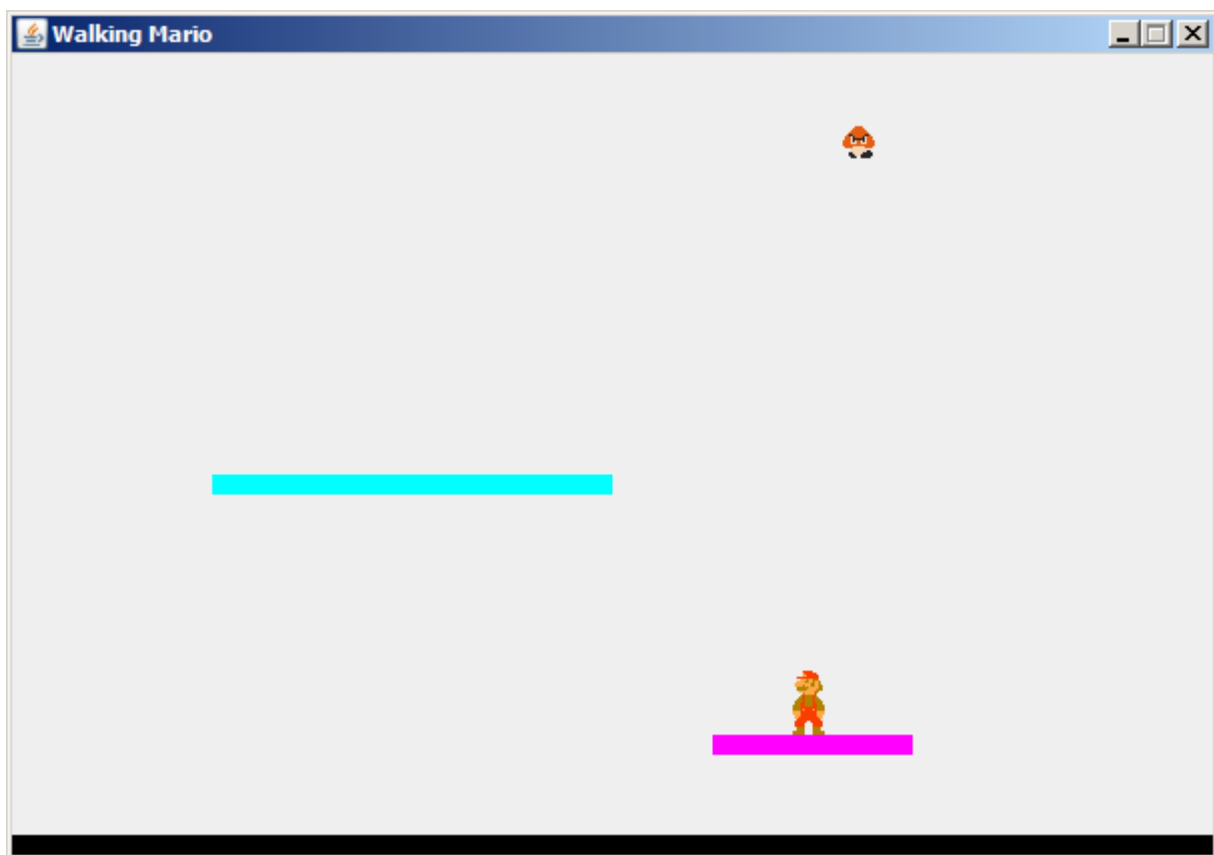
```
return my_rect.intersects(WalkingMario.plat.myRect());
```

... with this for loop:

```
for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
{
    if (my_rect.intersects(WalkingMario.plats.get(i).myRect()))
        return true;
}

return false;
```

Thankfully, we don't have to change any code in the complex `actuallyMove` method, because I designed that algorithm to work with any number of platforms! Run your program. **Try moving and jumping around on the platforms!** If you have done everything right, so far, your program's main window should look something like:



Try changing the positions and sizes of the platforms. Can you cause Mario to do anything "strange"? Now, we're going to fix various issues with Mario's "jumping", to make it more realistic (in the context of a Mario game). The first issue is that Mario can begin his jumps from anywhere. We want to make it so that Mario can only begin a jump, when he is on top of a platform.

Make sure you are in your `Mario.java` source file. Since the `act` method of the `Mario` class contains the code that causes Mario to respond to keyboard input, we probably need to make any necessary changes there. Right now, the `if` statement that controls Mario's response to the "jump" button looks like:

```
if (WalkingMario.ctrl.isJumpButtonDown())
{
    if (!jumping)
        startJump();
    else
        continueJump();
}
else if (jumping)
    stopJump();
```

If the "jump" button is currently down, and Mario is not currently in the middle of a jump, he will start a new jump, regardless of whether he is on top of a platform or not. We want to make it so that he can only start a new jump, when he is not currently in the middle of a jump, AND when he is on top of a platform. In order to do this, we need to specify that BOTH of these conditions must be true, in order for the `startJump` method to be called. In the above piece of code, change this block ...

```
if (!jumping)
    startJump();
```

... so that it looks like this:

```
if (!jumping)
{
    if (isOnAPlatform())
        startJump();
}
```

We need to use a nested `if` statement here, to ensure that if the "jump" button is down, Mario is not currently in the middle of a jump, AND Mario is NOT on top of a platform, NONE of his jumping-related methods will be called. **If students have trouble understanding why we're using a nested if statement here, guide them through the program logic.** Run your program. Now, Mario should only be able to begin a jump, when he is on top of a platform. Once you let go of the A key, Mario should begin to fall again, if he is not on top of a platform. If you press A, while Mario is in the air, he won't rise, as if he is jumping, but will continue to fall, until he lands on a platform. If you hold down the A key, while Mario is falling, he'll start a new jump, as soon as he lands on a platform.

The next issue we're going to fix is that Mario can jump as high as he wants. Mario is famous for his ability to jump high, but what goes up must come down, even Mario. Let's limit Mario's jumping power, so that he can only jump so high. We'll make some new member variables to store information about Mario's jumping power, so add the following member variables to your `Mario` class:

```
private int max_jumping_power;
private int jumping_power;
```

We'll limit Mario's jumping power, by specifying that the beginning and middle (combined) of one of his jumps will only last at most `max_jumping_power` ticks. If Mario is on top of a platform, the first tick following a press of the A key will involve a call to `startJump`. When our program calls that method, Mario will begin a jump with a full complement of jumping power, and then immediately expend one unit of that power. If the player continues to hold down the A key, our program will call the `continueJump` method. Each time our program calls that method, Mario will expend one unit of his jumping power, in the process of continuing to jump. We'll allow `continueJump` to be called up to `max_jumping_power - 1` times. If the player continues to hold down the A key, and Mario runs out of jumping power, the program will end Mario's jump. Once Mario's jump has ended (either by the player's releasing of the A key, or by Mario's running out of jumping power), the program will call the `stopJump` method, and Mario will begin to fall. Once Mario lands on a platform, his jumping power will be fully recharged to `max_jumping_power` units. From there, he will be able to begin another jump.

We'll limit the duration of Mario's jumps to 30 ticks, and start him off with a full complement of jumping power, so in the constructor of your `Mario` class, add the following lines of code:

```
max_jumping_power = 30;
jumping_power = max_jumping_power;
```

Now, we're going to make some small changes to the methods that control Mario's jumping behavior. First, we'll make our `startJump` method cause Mario to begin a new jump with a full complement of jumping power, and then expend one unit of that power. Add the following lines of code to your `startJump` method:

```
jumping_power = max_jumping_power;
jumping_power = jumping_power - 1;
```

Next, we'll make our `continueJump` method cause Mario to expend one unit of jumping power, whenever it's called. Add the following line of code to your `continueJump` method:

```
jumping_power = jumping_power - 1;
```

We won't need to make any changes to the `stopJump` method, because Mario won't expend any more jumping power, when he ends a jump. Now, let's recharge Mario's jumping power, whenever he lands on a platform. Go to your `respondToGravity` method. Its body should look like:

```
if (!jumping)
{
    if (!isOnAPlatform())
        y_velocity = gravity;
    else
        y_velocity = 0;
}
```

If we're not currently jumping, and Mario is on top of a platform, the code in the `else` block will be executed. We only want to modify the code in that block. Change ...

```
else
    y_velocity = 0;

... to:

else
{
    y_velocity = 0;
    jumping_power = max_jumping_power;
}
```

Now, we'll need to add a new rule to the list of rules governing when Mario can and can't perform the parts of his jumps. We implement these rules in our `act` method, so we'll need to change some of that method's code. Change the condition ...

```
if (WalkingMario.ctrl.isJumpButtonDown())

... to:

if (WalkingMario.ctrl.isJumpButtonDown() && jumping_power > 0)
```

Now, the if statement that controls Mario's response to the "jump" button should look like:

```
if (WalkingMario.ctrl.isJumpButtonDown() && jumping_power > 0)
{
    if (!jumping)
    {
        if (isOnAPlatform())
            startJump();
    }
    else
        continueJump();
}
else if (jumping)
    stopJump();
```

If the "jump" button is down, Mario will only begin or continue to jump, if he has the power to do so. If he has no jumping power left (even if the "jump" button is still down), his jump will be ended, by a call to the `stopJump` method.

Run your program. Now, if you hold down the A key, Mario will only jump so high. Once Mario has used up all of his jumping power, he won't be able to rise again, until he lands on a platform. If you hold down the A key for a long enough time, Mario will just keep on jumping: boing, boing, boing

However, there is yet another issue, with Mario's jumping, that we need to fix. **Go underneath the magenta platform. Press the A key, and hold it down for a while.** Notice how once Mario hits his head on the underside of the platform, he keeps trying to rise? If you're also trying to move left or right, at the same time, he "walks" in that direction, but in place, as if he's blocked by an invisible wall.

This is a consequence of our collision-detection algorithm. When Mario hits a platform, while rising, his movement is interrupted. If he still has jumping power, though, he keeps trying to rise, until he runs out. In a Mario game, when Mario hits his head on a block, he immediately begins to fall again, rather than pushing his head up against the underside of that block for a noticeable amount of time. If Mario hits his head on the underside of a platform, while jumping, we want to end his jump right there, and cause him to start to fall again.

First, we'll need a method, similar to `isOnAPlatform`, that will check if Mario's head is right under the underside of a platform. His head is in such a location, when BOTH of the following are true:

- The top of Mario's rectangle is exactly one pixel below the bottom of the platform's rectangle.
- The horizontal extents of Mario and the platform share at least one pixel.

Add the following method to your Mario class:

```
public boolean isHeadUnderAPlatform()
{
    for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
    {
        Platform plat = WalkingMario.plats.get(i);

        if (myTop() == plat.myBottom() + 1 &&
            horizontalExtent().overlaps(plat.horizontalExtent()))
            return true;
    }

    return false;
}
```

Then, since the interruption of Mario's upward movement is caused by code in our `actuallyMove` method, we'll need to make a small change there. If Mario hits a platform, from any direction, the `actuallyMove` method will undo his last "sub-move", and then break out of that for loop. Between the undoing of this "sub-move" and the breaking out of the for loop, we'll check if Mario hit his head on the underside of a platform. If he hit his head, we'll take away his jumping power. In the `actuallyMove` method, right before the break statement, which looks like:

```
break;
```

Insert the following if statement:

```
if (isHeadUnderAPlatform())
    jumping_power = 0;
```

If Mario hits his head, he will lose all of his jumping power. On the next tick, when his `act` method is called, the condition, `WalkingMario.ctrl.isJumpButtonDown() && jumping_power > 0`, will be `false`, because his jumping power will not be greater than 0. Therefore, the `stopJump` method will be called, to end his jump, because he has to be in the middle of a jump (jumping has to be `true`) to hit his head.

Run your program. **Go underneath the magenta platform again, and hold down the A key for a while.** When Mario's head hits the underside of that platform, he should immediately start to fall again. If you keep holding down the A key, Mario will land on the black platform, jump, hit his head on the magenta platform, fall, land on the black platform, and so on: boing, boing, boing

Now, Mario can jump correctly! However, there's still one last issue we need to fix. Run your program, and try pausing the action, by pressing the P key. When you pause it, the word, `PAUSED`, will appear in magenta. It will appear in magenta, because when `PAUSED` is drawn on the screen, magenta is the current color. Go to your `ContentPane.java` source file. Look at the following code, in the body of the `paintComponent` method of your `ContentPane` class:

```
for (int i = 0; i < WalkingMario.plats.size(); i = i + 1)
    WalkingMario.plats.get(i).draw(g);

...

if (WalkingMario.paused)
    g.drawString("PAUSED", 160, 140);
```

First, we draw each of our platforms. Each time we draw a platform, we change `g`'s current color to that platform's color. Our three platforms are black, cyan, and magenta, so we end up changing `g`'s current color to black, cyan, and magenta, in that order. When it's time to draw `PAUSED` on the screen, `g`'s current color is still magenta, so `PAUSED` gets drawn in magenta. If the program is paused, let's draw the word, `PAUSED`, in black, rather than in the color of the last platform we added to the world. Change ...

```
if (WalkingMario.paused)
    g.drawString("PAUSED", 160, 140);

... to:

if (WalkingMario.paused)
{
    g.setColor(Color.BLACK);
    g.drawString("PAUSED", 160, 140);
}
```

Make sure you import the `Color` class, with the following import statement:

```
import java.awt.Color;
```

Run your program. If you pause it, the word, `PAUSED`, should always be black, rather than the color of the last platform you added to your game's world. **Try changing various things about Mario, such as his jump strength and maximum jumping power. Try re-arranging the platforms in Mario's world. Challenge yourself, by creating gaps between platforms that require "the perfect jump" to cross without falling into the abyss.** Before you change any code, you might want to save a backup copy first, in case you mess something up, and don't know how to fix it.